Using Objects in THINK C 23

This chapter shows you how to use the object-oriented extensions to THINK C. The THINK C implementation of objects is derived from C++ as described in *The Annotated C++ Reference Manual* by Margaret Ellis and Bjarne Stroustrup.

The object extensions to THINK C are based on C++ but don't include some features of the language, including multiple inheritance and operator overloading. For a complete implementation of C++, use Symantec C++, described in Chapter 1, "Welcome."

How to read this chapter

This is a long chapter, containing a lot of information. Don't feel that you have to understand it all at once. Read a little at a time. Try out the examples. The LearnOOP program in your THINK C package demonstrates many of the new features in a fairly small program.

Contents

Introduction 425
    What you should know 425
    A note on terminology 425

Using the Object Extensions 426

Overview 426

Declaring a Class 428
    Declaring root classes 429
    Declaring subclasses 430

Declaring and Using Objects 430
    Creating and deleting objects 430
    Assignment and objects 432
    Objects and memory 433

Accessing Members of a Class 433

Declaring Members of a Class 435
Declaring instance variables 435
Declaring class variables. 437
Declaring methods 438
Virtual and nonvirtual methods 438
Class methods 439
Examples of methods. 439
Declaring circular and forward references 440

Defining and Using Methods. 441

Access Control 443
Using access specifiers 444
Access specifiers in subclass declarations 445
Protected access 446

Constructors and Destructors 448
Declaring and defining constructors 448
How constructors are invoked. 449
Using access specifiers with constructors 450
Destructors 450

Allocators and Deallocators 451
Allocators. 452
Deallocators 453
Default allocator and deallocator 454

Tips and Techniques 454
Organizing your classes 455
Objects and the Macintosh 455
Direct and indirect classes 458

Object Extension Options. 458
Turning on the object extensions 458
Classes are indirect by default 459
Methods are virtual by default 460
Generate class names. 460
Optimize monomorphic methods. 460

Special Functions 460

Differences Between THINK C and C++ 461
General differences 461
Keywords. 462
Value objects and reference objects 462
THINK C extensions 463
Access declarations, friends, and multiple inheritance . . . 463
Constructors and destructors 463
Scope resolution operator 463
Sizeof operator not allowed 463

Introduction

This chapter describes how THINK C implements objects. Implementation is based largely on Object Pascal and borrows its syntax from C++. It also adopts other features of C++ such as access control, constructors and destructors, and the `class` keyword.

This chapter begins with a section that tells you how to turn on the object extensions to THINK C and what libraries and headers you need. Next, find a quick overview on defining a class, creating an object, defining a method, and so on. The sections that follow go into more detail on each aspect of the object extensions.

What you should know

Before you read this chapter, you should know the basics of object-oriented programming. In particular, you should know about classes, instances, instance variables, and methods. If you need to learn about these things, see Chapter 22, "Object-Oriented Programming." You should also be familiar with the Macintosh Memory Manager. To learn about it, see *Inside Macintosh II*, Chapter 1, "The Memory Manager."

A note on terminology

Though the implementation of objects in THINK C is based on C++, most of the terminology comes from Smalltalk and Object Pascal. If you read the introduction to object-oriented programming earlier in this manual, or if you've used objects in THINK C before, the terminology in this chapter will be familiar. If you know C++, the examples will look familiar, but the terminology might be foreign to you. Here's a short glossary for clarification.

THINK C Term	C++ Term
method	member function, function member
instance variable	data member
superclass	base class
subclass	derived class
class method	static member function
class variable	static data member

Table 23-1 THINK C terms

Note

This chapter uses the word member to mean instance variable or method.

Using the Object Extensions

See "Object Extension Options" later in this chapter, for information about options that affect the object extensions.

To use the object-oriented programming extensions to THINK C, you must make sure you select the THINK C + Objects option in the Language Settings page of the THINK C options dialog.

Every program that uses objects in THINK C needs one of the libraries in the `oops Libraries` folder. The particular library you need depends on the kind of application you're using.

If you're writing…	You need this library
An application	`oops`, `oopsDebug`
An application with Far Code	`oopsFar`
An application with Far Data	`oopsFar`
A code resource	`oopsA4`
A desk accessory	`oopsA4`
A driver	`oopsA4`

Table 23-2 The appropriate object-oriented library for a project

The library called oopsDebug is designed to catch certain common errors that might occur while you're developing object-oriented applications. This library works only with applications that don't use Far Code or Far Data. If you want, you can make a copy of oopsDebug and rebuild it with the Far Code and Far Data options on.

For a description of these functions see "Special Functions" later in this chapter.

Any source file that uses the special functions `member()`, `new_by_name()`, or `class_name()` must include `<oops.h>`.

Overview

This section is a quick introduction to using the object extensions in THINK C. It gives you a broad picture of each step involved: declaring a class, creating an object, referring to instance variables and methods, and defining the methods of a class. The purpose of this section is to give you a working vocabulary. The sections that follow build on what you learn here.

To use objects, you first need to declare a class. A class declaration describes the properties of a class and names the instance variables

and methods. A class declaration is very much like a `struct` declaration.

```
struct MyClass : indirect {
  int  a;           // an instance variable
  int  b;           // an instance variable
  void M(int);      // a method
                    // declaration
  int  N(int, int); // a method
                    // declaration
};
```

To use an object, declare an object reference—a pointer to an object—then use the `new` operator to allocate space for it. This is how you declare an object reference:

```
MyClass *anObject;   // OK: anObject is an
                     // object reference
```

But you can't do it like this:

```
MyClass anObject;    // ERROR: illegal use
                     // of class type
```

Since `anObject` is a pointer, you need to allocate space in memory before you can use it. For objects, use `new` like this:

```
anObject = new MyClass;
```

Some books on object-oriented programming refer to objects as "instances" of a class and to creation of objects as "instantiation."

The `new` operator creates a new object in your program's heap. In THINK C, most objects are allocated as handles, though you can create objects as pointers.

Now you can use the members—the instance variables and methods—of the object. To use an instance variable, use the same syntax that you use to refer to the field of a `struct`. This example shows how to change the values of the instance variables `a` and `b`:

```
anObject->a = 10;
anObject->b = anObject->a++;
```

You use a similar syntax to call a method. This example sends an `M()` message to the object `anObject`:

```
anObject->M(6);
```

Of course, if you don't define the method, you get a link error when you try to run your program. Defining a method is almost like

defining a function. To define a method `M()` for a class `C`, name the function `C::M`. So the definition of the method `M()` in the example above might look like this:

```
void MyClass::M(int arg)
{
   int  t;
   int  b;

   t = N(arg, a);     // N is a member
                      //   function
                      // a is an instance
                      //   variable
   b = t * 2;         // b is the local
                      //   variable
   this->b = t * 2;   // this->b is the
                      //   instance variable
}
```

Some special rules apply in a method definition. Whenever you define a method, it's as if this variable definition appeared at the beginning of the method:

```
register MyClass *this;
```

`this` is the current object—the object that received the message. When you use the name of a member in a method, it is as if you had written `this->` before the name. In the example above, the call to `N()` is the same as sending the message `N()` to `this`. If there's a local variable with the same name as an instance variable, you need to use `this->` explicitly to refer to the instance variable. In the example above, `b` is a local variable, so to refer to the instance variable you need to write `this->b`.

The next sections detail each aspect of using objects: class declaration, method and instance variable declaration, class creation, method definition, and inheritance. Other sections cover topics that this quick introduction does not touch on: access control, constructors and destructors, and allocators and deallocators.

Declaring a Class

A class declaration describes the representation of objects and the operations they can perform. You can use the `class` keyword or

the `struct` keyword to declare a class. When you declare a class, the name of the class becomes a type.

Declaring root classes

A root class is a one that does not inherit any members from any other class. The root class specifies the representation for a family of classes and whether references to members are made through a pointer or through a handle.

This is what root class declarations look like:

```
class class-name {
    member-declarations
};

class class-name : storage-specifier {
    member-declarations
};

struct class-name : storage-specifier {
    member-declarations
};
```

See "Direct and indirect classes" later in this chapter for more information.

The storage specifier is either `direct` or `indirect`. If the storage specifier is `direct`, THINK C assumes that objects of the declared class are allocated as pointers and only does a single dereference to access members of those objects. If the storage specifier is `indirect`, THINK C assumes that they're allocated as handles and does a double dereference to access members.

See "Object Extension Options" later in this chapter for information about options that affect the object extensions.

For classes declared with the `class` keyword, the storage specifier is optional. If the Classes Are Indirect by Default option is on, objects of such a class are `indirect`; otherwise they're `direct` objects.

The member declarations describe the instance variables and methods that objects of the class will have.

Access specifiers are described in the section "Access Control" later in this chapter.

When you use the `class` keyword, the members of the class—all the instance variables and methods—are implicitly `private`. When you use the `struct` keyword, the members of the class are implicitly `public`.

Declaring subclasses

A subclass is a class that inherits members from a root class or from another subclass. This is what a subclass declaration looks like:

```
class class-name : access-specifier superclass {
    member-declarations
};

struct class-name : access-specifier superclass {
    member-declarations
};
```

For details and examples of access specifiers to superclasses, look at "Access specifiers in subclass declarations" later in this chapter.

The access specifier determines how members of the superclass are treated in the subclass. If the access specifier is `public`, `public` and `protected` members of the superclass are treated as `public` and `protected` in the subclass. If the access specifier is `private`, the `public` and `protected` members of the superclass are treated as `private` members of the subclass.

If the access specifier is missing, it defaults to `private` for classes declared `class`, and `public` for classes declared `struct`.

Declaring and Using Objects

Unlike C++, THINK C does not allow you to declare objects that are not pointers.

Once you've defined a class, you can use its name to declare or define an object reference. An object reference is always declared as a pointer to a value of class type. This is how you declare an object:

```
class-name *object-reference;
```

Warning

Objects are always declared as pointers, regardless of whether the storage specifier was `direct` or `indirect`. If you leave out the `*`, you get the error `illegal use of class type`.

An object reference definition allocates memory for the object reference only, not for the object. To allocate the memory for the object, use the `new` operator, described in "Allocators and Deallocators" later in this chapter.

Creating and deleting objects

Before you can use an object, you must allocate space for it in the heap. When you're finished with an object, you should delete it to

reclaim its space. The operators `new` and `delete` create and delete objects.

To create an object, use `new` like this:

object-reference = `new` *class-name*;

To delete an existing object, use `delete` like this:

`delete` *object-reference*;

Note

`new` and `delete` are keywords supported by the compiler when the THINK C + Objects option is on.

Here is an example:

```
struct A : indirect {
    member declarations
};

struct B : A {
    member declarations
};

void f()
{
    A  *a;
    B  *b;

    b = new B;
    a = new B;

    delete a;
    delete b;
}
```

Example 23-1 Creating and deleting objects

Virtual methods are described in "Defining and Using Methods" later in this chapter.

In Example 23-1, note that `a` is declared as being of type `A`, but created as an object of class `B`. This means that `a` has a declared type of `A`, but a run-time type of `B`. When you send a message to `a`, and the message refers to a virtual method, it is treated as if it were of class `B`.

Assignment and objects

You can assign one object reference to another if both objects are of the same class, or if you're assigning an object of a subclass to an object of a superclass and the subclass was publicly derived from the

superclass. The rules for object assignment are also used to match objects against the arguments of functions.

Following are some examples:

```
class A : indirect {
   member declarations
};

class B : public A {
   member declarations
};

class C : public B {
   member declarations
};

class K : A {
   member declarations
};

class L : K {
   member declarations
};

void g(A *x) { ... }

void f(void)
{
   A *a;
   B *b;
   C *c;
   K *k;
   L *l;

   a = c;   // OK: C is public
subclass
            //   of A
   a = k;   // ERROR: K is private
subclass
            //   of A
```

Example 23-2 Legal and illegal object assignments

```
   g(b);    // OK: B is public
subclass
            //   of A
   g(l);    // ERROR: L is private
subclass
            //   of A

   a = new B;  // OK: B is public
subclass
               //   of A
   a = new K;  // ERROR: K private
subclass
               //   of A
}
```

Example 23-2 Legal and illegal object assignments *(Continued)*

Objects and memory

See "Object Extension Options" later in this chapter, for information about options that affect the object extensions.

If a class is declared `indirect` (or if the Classes Are Indirect by Default option is on and the class is declared without a storage specifier), THINK C allocates an object of that class as a handle. If a class is declared `direct`, an object of that class is allocated as a pointer. In both cases, object references are declared as pointers. If the space is allocated as a handle, THINK C does the additional dereference implicitly.

THINK C uses the Toolbox routines `NewHandle` and `NewPtr` to allocate space for objects, so you can use Memory Manager routines that work on pointers and handles. For instance, if you have an object of an `indirect` class, you might want to lock it down while you call other Toolbox routines that move memory.

You can define allocators and deallocators for a class if you want to allocate memory a different way. See the section "Allocators and Deallocators" later in this chapter for more information.

Accessing Members of a Class

If you have an object reference, and you want to access one of its class members, use the same syntax that you use to refer to a member of a `struct` through a pointer. The form for accessing a member of a class is:

object-reference->member-name

The object reference is an object created with new. The member name is the name of the member that you want to access. If the member is not in the same class as the object reference, THINK C looks in the superclass.

The exact syntax for declaring members of a class is given in the next section. Many of the examples in this chapter access members of classes, so you should be familiar with the basic form. Come back to this section later to learn the details.

In Example 23-3, assume that class B is a descendant of class A—in other words, A is B's superclass. Assume that m is an instance variable declared in B and that n is an instance variable declared in A.

```
struct A : indirect {
   short n;
   other member declarations
};

struct B : A {
   short m;
   other member declarations
};

void g(void)
{
   B  *b;

   b = new B;  // b is an object of B
   b->m = 1;   // m is a member of B
   b->n = 1;   // n is a member of A
               //   inherited by B
   delete b;
}
```

Example 23-3 Accessing members of a class

If the member you want to access is a method, use the same syntax, but as if you were calling a function:

```
void g(void)
{
   B  *b;

   b = new B;  // b is an object of B
   b->Meth();  // Meth is a
               //   method of B or
}              //   inherited by B
```

Example 23-4 Calling a method

When you access an instance variable or call a method, THINK C starts looking for the member in the run-time class of the object

reference. You can use the scope resolution operator `::` (two colons) to specify in which class the search should begin. The more general form of member access is:

object-reference`->`*class-name*`::`*member-name*

Virtual methods are described in "Defining and Using Methods" later in this chapter.

For example, suppose that B and A both declare a virtual method `M()` and that A is a superclass of B. You can use the scope resolution operator to bypass the inheritance rules like this:

```
void g(void)
{
   B  *b = new B;
   A  *a;

   a = new B;  // a's run time class is B

   b->M();     // call B's M()
   a->M();     // since M() is virtual,
               //   calls B's M()
   a->A::M();  // calls A's M()
}
```

Example 23-5 Using the scope resolution operator

Declaring Members of a Class

To learn about access specifiers see "Access Control" later in this chapter.

In THINK C, members of a class must be either data or functions. A class can declare any number of instance variables and must declare at least one method. Access specifiers may be placed wherever they're needed within the member declarations.

Declaring instance variables

The declaration of an instance variable in a class is similar to the declaration of a field in a `struct`.

```
struct A : indirect {
   int   a;
   int   b;
   at least one method declaration
};

class B : public A {
   int   c;
   int   d;
   at least one method declaration
};
```

Example 23-6 Declaring instance variables.

A subclass inherits all the instance variables of the superclass, as if they had been declared in the subclass. In Example 23-6, class B inherits the instance variables a and b from class A. It is as if class B had been declared like this:

```
struct B : indirect {
   int   a;
   int   b;
   int   c;
   int   d;
   at least one method declaration
};
```

Example 23-7 The effect of inheritance

If you declare an instance variable with the same name as an instance variable in a superclass, two separate instance variables exist. An object of the subclass has two different instance variables with the same name.

```
struct A : indirect {
   int   a;
   int   b;
   at least one method declaration
};

struct B : A {
   long  b;
   int   m;
   at least one method declaration
};
```

Example 23-8 Instance variables with the same name

In Example 23-8 class A has an instance variable named b. Class B, a descendant of class A, has another instance variable named b. If you create an object of class B and access the instance variable b, you will be referring to the b defined in class B. You can use the scope resolution operator to refer to the instance variable in the superclass.

```
void F (A *p, B *q)
{
   p->b = 10;     // This is A::b
   q->b = 10;     // This is B::b
   q->A::b = 10;  // This is A::b
}
```

Example 23-9 Referring to a specific instance variable

Declaring class variables

An instance variable declared `static` is a class variable. A class variable isn't part of an object of its class. Instead, all the objects of the class share the class variable. You might want to use a class variable to keep information that pertains to all the objects of the class. For example, for a class that represents some limited resource, you can use a class variable to keep track of its use.

Before you can use a class variable, you need to define it. Its declaration in the class declaration does not allocate space for it.

```
struct A : indirect {
   static int count;    // class variable

   int   a;
   int   b;

   void  F(void);
};

int A::count = 0;
   // defining and initializing
   // the class variable
```

Example 23-10 Declaring and defining a class variable

Unlike other members of a class, you don't need an object reference to refer to a class variable. If you do provide an object reference, it is used to determine the class. So the statements below all refer to the same class variable:

```
void g(A *a)
{
   A::count = 10; // set count to 10
   a->count = 20; // same as A::count = 20
}
```

Example 23-11 Accessing a class variable

To learn about access control, see "Access Control" later in this chapter.

The access control rules apply to class variables. If the class variable `count` in Example 23-10 had been declared `private`, only methods of class `A` could access and modify it. The only exception to the access control rules is that you are always allowed to initialize a class variable in global scope.

Declaring methods

See "Object Extension Options" later in this chapter for information about options that affect the object extensions.

The syntax for declaring a method is the same as for declaring a function. You can declare a method normally or declare it to be `virtual` or `static`. If the methods are virtual by default option is on, all nonstatic method declarations are implicitly `virtual`.

```
struct A : indirect {
   int   a;
   int   M(int);         // normal
                         //  declaration
   virtual int V(int);   // virtual
                         //  declaration
   static int S(int);    // static
};                       //   declaration
```

Example 23-12 Method declarations in a class

In THINK C, every class—whether declared `struct` or `class`—must have at least one method.

A method declared in a subclass can have the same name as a method declared in its superclass. If a method is declared `virtual` in a superclass, a method of the same name in a subclass is also `virtual`, whether or not the method in the subclass is declared `virtual`. The return types and argument lists for virtual methods must match between superclass and subclass.

Virtual and nonvirtual methods

Virtual methods are the mechanism that makes object-oriented programming work. When you send an object a message—that is, when you invoke one of its methods—the actual method that executes depends on the run-time class of the object. When you declare a method in a subclass for a virtual method that exists in the superclass, you are overriding the method.

If a method in a superclass is declared nonvirtual (and the Methods Are Virtual by Default option is off), you can declare a method with the same name in a subclass. The two methods may have different return types and argument lists. Nonvirtual methods in a subclass do not override a method in the superclass. There simply are two methods with the same name.

Class methods

A method declared `static` is a class method that doesn't behave like a nonvirtual or virtual method. Instead, it's a function whose scope is limited to the class that declares it and to the subclasses that have access to it. Class methods are useful as auxiliary routines. In Macintosh programs, you can use a class method declared `pascal` as a callback routine.

Note

You cannot take the address of a virtual or nonvirtual method, but you can take the address of a class method.

Examples of methods

The use of `virtual` *here is optional, but it is good programming style.*

The following example illustrates the different method declarations and method invocations:

```
struct A : indirect {
   virtual  void V1(void);
   virtual  void V2(void);
            void M1(void);
   static   void S1(void);
};

struct B : A {
   virtual  void V1(void);
            void M1(int);
};


void g(void)
{
   A  *a;      //declared type is A *
   B  *b;

   a = new B;  // run-time class of
               // a is B
```

Example 23-13 Method declarations and invocations

```
    b = new B;

    a->V1();    // invokes B::V1()
    b->V1();    // invokes B::V1()
    b->V2();    // invokes A::V2()
    a->M1();    // invokes A::M1()
    b->M1(1);   // invokes B::M1()
    b->A::M1(); // invokes A::M1()
    b->S1();    // invokes A::S1()
    A::S1();    // invokes A::S1()
}
```

Example 23-13 Method declarations and invocations *(Continued)*

When you call a virtual method, the THINK C message dispatcher looks at the run-time class of the object and invokes the appropriate method. In Example 23-13 above, the run-time class of both `a` and `b` is class `B`, so when `V1()` is called, class `B`'s `V1()` is invoked.

If the method is monomorphic and the Optimize Monomorphic methods option is on, the linker bypasses the dispatcher and generates a call directly to the method. A monomorphic method is a method that does not override another method and is never overridden.

Declaring circular and forward references

Because the name of a class is known by the time you declare members of that class, you can use its name for circular references. For example:

```
struct AList : indirect {
   int   someInfo;
   AList *nextItem;
   method-declarations
};
```

Example 23-14 A circular reference.

If a class declaration needs to refer to another class that hasn't been declared yet, you can use the `class` keyword to name the class before you use it:

```
class B;       // name the class before
               // using it

struct A : indirect {int a;void F(B *b);
};
```

```
struct B : indirect {
   int   m;
   void  G(A *a);
};

g (A *p, B *q)
{
   p->F(q);    // now it works
}
```

Example 23-15 A forward reference

Use the same technique for declaring classes that are mutually dependent:

```
class B;       // name the class before
               //  using it
struct A : indirect {
   B     *anObject;
   void  F(void);
};

struct B : indirect {
   A     *anObject2; // A is already
                     //   declared
   void  G(void);
};
```

Example 23-16 Mutually dependent classes

Defining and Using Methods

You define a method the same way you define a function, except that you include the class name before the method name, like this:

return-type class-name::*member-function*(*parameter-decls*)
{
    function-body
}

Within a method, the compiler supplies an implicit declaration so you can refer to the object that received the message:

```
register class-name *this;
```

The symbol `this` *is analogous to the implicit variable* `self` *in Object Pascal or Smalltalk.*

The symbol `this` refers to the object receiving the message. Within a method definition, you can omit `this->` when you refer

to an instance variable or when you call a method in the same class or superclass as `this`.

If your method definition redeclares a variable with the same name as an instance variable or method, you can use `this->` or the scope resolution operator to recover the original meaning.

```
struct A : indirect {
   int   a;
   int   b;
   void  M(void);
};

void A::M(void)

{
   int   a;       // local variable with
                  // same name as an
                  // instance variable

   b = 7;         // sets instance
                  //   variable to 7
   a = 5;         // sets local
                  //   variable to 5
   this->a = a;   // sets instance
                  //   variable to 5
   A::a = a;      // sets instance
}                 //   variable to 5
```

Example 23-17 Local and instance variables with the same name

Within a method definition, you may want to use the superclass's method. For example, to add functionality to a method, use the superclass's method, then add more code to augment the behavior.

This is how you call a superclass's method:

superclass`::`*member-function*`(`*args*`)`

To refer to the immediate superclass of the current class, use the word `inherited`, like this:

`inherited::`*member-function*`(`*args*`)`

When you use the scope resolution operator to invoke a method, THINK C bypasses the message dispatcher and generates a direct call.

Note

This use of the word `inherited` is a THINK C extension and is not found in standard C++. However, it is supported in Symantec C++.

Access Control

THINK C implements an access control mechanism based on C++. Access control lets you specify those instance variables and methods available to subclasses and other classes. You can use the access control mechanism to make sure that only subclasses change the values of instance variables or to ensure that certain methods are available only to the class that defines them.

An instance variable or method can be `public`, `private`, or `protected`.

Access Control Labels	Meaning
`public`	The method or instance variable is available anywhere.
`private`	The instance variable or method is available only to methods declared in the class that the declaration appears in.
`protected`	The instance variable or method is available only to methods declared in the class that the declaration appears in or in subclasses of that class.

Table 23-3 Access control labels

Note

The words `public`, `private`, and `protected` are not keywords in THINK C as they are in C++. THINK C interprets the access specifiers only in class declarations.

The way you declare a class—whether or not you use the `class` or `struct` keyword—affects the default access control for that class.

Classes declared with…	Have this access control
`class`	private
`struct`	public

Table 23-4 Default access control

Using access specifiers

An access specifier—`public`, `private`, or `protected`—can appear before any instance variable or method declaration in a class declaration. The access specifier is in effect until the next specifier or until the end of the class declaration.

Following are some examples:

```
class ClassA {
   int   a;          // a is private
                     //   by default
private:
   int   b;          // b & c are private
   int   c;
protected:
   int   d;          // d is protected
public:
   int   E(void);    // E() is a public
                     //   method
private:
   int   F(int x);   // F() & g are private
   int   g;
};
```

Example 23-18 Access specifiers for members of a class

Remember that when you use the `class` keyword to declare a class, the members of that class are `private` by default. If you use `struct` to declare a class, all methods are public by default:

```
struct ClassB : indirect {
   int   h;          // h is public
   int   M(int x);   // so is M()
private:
   int   i;          // but i and j
   char  j;          // are private
};
```

In these examples, the instance variables `ClassA::a`, `ClassA::b`, `ClassA::c`, `ClassA::g`, and the method `ClassA::F()` are all private. The only places that you would be allowed to refer to them are in the methods `ClassA::E()` and `ClassA::F()`. Since `E()` is the only public method of `ClassA`, that's the only message you can send to an object of `ClassA`.

Access specifiers in subclass declarations

When you create a new class, you can use the `public` and `private` access specifiers to control access to the instance variables and methods of the superclass.

If you specify that the superclass should be `public`, then all the `public` and `protected` members of the superclass will be `public` and `protected` in the subclass. If you specify that the superclass should be `private`, the `public` and `protected` members of the superclass will be private members of the subclass.

If you don't provide an access control specifier, THINK C assumes `private` when the subclass is declared with `class`, and `public` when the subclass is declared with `struct`.

This example shows what happens when you try to access a class declared `private` in another function.

```
class A {      // a root class
   int   a;
public:
   int   b;
   void F(void);
};

class B1 : public A {
   int   m;
public:
   int   n;
   void  G(void);
};

class B2 : private A {
   int   w;
public:
   int   x;
   void  Q(void);
};
```

Example 23-19 Access specifiers for superclasses

```
void fn(void)
{
   B1    *p = new B1;
   B2    *q = new B2;

   p->a;       // ERROR: A::a is
private
   p->b;       // OK: A::b is public
to B1
   q->b;       // ERROR: A is
private to B1
}
```

Example 23-19 Access specifiers for superclasses *(Continued)*

Protected access

Protected access lets subclasses access members of a superclass without making those members public. For example, in an abstract class you might want to make certain implementation-level details available only to subclasses of that class, not to other classes.

The rules for protected access are not as intuitive as are those for private and public access. Access to a protected member is restricted not only by the class of the object reference but also by the class of the method the reference appears in.

To refer to a protected member, three conditions must be true:

1. The reference must appear in a method definition.

2. The method in which the reference appears must be of the same class as the class in which the protected member is declared (or in a subclass of that class).

3. The object through which the reference is made must be in the same class as the method in which the reference appears (or in a subclass of that class).

The examples below assume these declarations:

```
struct A : indirect {
protected:
   int   a;
   void  M1(void);
};

struct B1 : A {
   void  M2(void);
};

struct B2 : A {
   void  M3(B1 *q);
};
```

In Example 23-20, the method that the protected member `a` is referenced in is of the same class as `a`. Since the reference is through `this->`, which is of the same class as `M1()`, the instance variable `a` is accessible.

```
void A::M1(void)
{
   a = 1;   // ok
}
```

Example 23-20 Protected access through the same class

In Example 23-21 the method that the reference appears in is defined in a subclass of the class that `a` is declared in. In other words, `B1` is a subclass of `A`. As in the previous case, the reference is through `this->`. Because `this` is of the same class as the method, `B1`, the instance variable `a` is accessible.

```
void B1::M2(void)
{
   a = 1;   // ok
}
```

Example 23-21 Protected access through a subclass

In Example 23-22 `M3()` is of class `B2`, which is a subclass of `A`, the class the protected member is declared in. The reference, though, is through `q`, which is of class `B1`. Since `q` is not the same

class as the method, B2, the protected instance variable is not accessible.

```
void B2::M3(B1 *q)
{
   q->a = 1;   // illegal
}
```

Example 23-22 Illegal protected access

Constructors and Destructors

A class can have two special methods that are called when an object is created and destroyed. The method that's called when an object is created is called a constructor, and the method called when an object is deleted is called a destructor.

Declaring and defining constructors

In THINK C, you can't call a constructor directly to create an object, as you can in C++.

A constructor is a method that has the same name as the class it's declared in. For example, the constructor for the class SomeClass is named SomeClass::SomeClass(). A constructor is invoked automatically when you create an object with new. If an object has a superclass, the superclass's constructor is called before the current class's constructor is called.

There are some special rules for declaring a constructor. You cannot declare a constructor virtual or static. You cannot specify a return type for a constructor—not even void. This is how you declare a constructor:

```
class SomeClass {
   int            theLastCount;
   AnotherClass   *list;
   SomeClass(void);
   virtual int GetCount();
   ...
};
```

Example 23-23 Declaring a constructor

The same rules apply when you define the constructor.

```
SomeClass::SomeClass (void)
{
   theLastCount = GetCount;

   list = new AnotherClass;
}
```

Example 23-24 Defining a constructor

How constructors are invoked

If a class is not a root class, the superclass's constructors are called from the root class to the class of the object being created. For example, suppose you have these classes:

```
class A {
public:
   A(void);
   virtual void F(void);
};

class B : A {
public:
   B(void);
   virtual void F(void);
};

class C : B {
public:
   C(void);
   virtual void F(void);
};

void f(void)
{
   C  *theC;

   theC = new C;
}
```

Example 23-25 Constructors in a class hierarchy

When you create an object of class `C`, the constructors are called in order: `A()`, `B()`, and `C()`, the way you would expect. One thing you might not expect is that while `A()` is running, the class of the current object `this` is `A`, while `B()` is running `this` is of class `B`, and it's not until `C()` executes that `this` is of class `C`. The implication of this behavior is that if your constructors call the

method F(), a different F() is invoked. In A(), A::F() executes; in B(), B::F() executes; and in C, C::F() executes.

Using access specifiers with constructors

You can use access specifiers to control access to constructors and destructors. If you declare the constructor of class A protected, only subclasses of class A are allowed to create those objects. If you declare a constructor private, the only way you can create an object of that class is to do so in a class function. That's why the constructors in Example 23-25 are declared public.

Destructors

A destructor is the opposite of a constructor. When you delete an object, its destructor is invoked. If the object has a superclass, its destructor is invoked, and so on until the destructor of the root class is reached.

Similar rules about declaring constructors apply to destructors. A destructor cannot be static, and a destructor cannot have a return type. In C++ a destructor can be declared virtual. In THINK C, a destructor is *always* virtual. Below is an example of how you declare a destructor:

```
class SomeClass {
   int            theLastCount;
   AnotherClass   *list;
   SomeClass(void);        // Constructor
   ~SomeClass(void);       // Destructor
   virtual int GetCount();
   ...
};
```

Example 23-26 Declaring a destructor

And this is how you define a destructor:

```
SomeClass::~SomeClass(void)
{
   delete list;
}
```

Example 23-27 Defining a destructor

The destructor invoked depends on the run-time type of the object being deleted. Consider this example:

```
class A {
public:
   ~A(void);
};

class B : A {
public:
   ~B(void);
};

class C : B {
public:
   ~C(void);
   virtual F()
}

f ()
{
   A  *theThing = new C;
   ...
   delete theThing;
}
```

Example 23-28 Destructors in a class hierarchy

In this example, `theThing` is declared as being of class `A`, but its run-time type is class `C`. When `theThing` is deleted, the destructor invoked is `~C` not `~A`. Destructors are called in this order: `~C()`, `~B()`, `~A()`. As destructors are invoked, the type of the object changes as it's destroyed. In `~C()`, the class of `this` is `C`, in `~B()` it's `B`; in `~A()` it's `A`.

Warning

This behavior matches the C++ behavior for virtual destructors. For maximum compatibility with C++, be sure to declare destructors `virtual`.

Allocators and Deallocators

Allocators and deallocators let you customize how memory is allocated for an object. An allocator allocates memory for the object, and a deallocator returns an object's memory to the free store. Allocators execute before any constructor, and deallocators execute after the last destructor.

Allocators and deallocators are always `static` and have `public` access regardless of how they're actually declared.

Allocators

Allocators allocate memory for an object. If `C` is the name of a class, the name of its allocator is `C::operator new()`. The function `operator new` takes the number of bytes to allocate as a parameter and returns a pointer to the allocated memory:

The type `size_t` *is defined in* `<stddef.h>`*. THINK C lets you use* `long` *or* `unsigned long` *instead of* `size_t`.

```
class C {
   void * operator new (size_t);
   ...
};
```

Example 23-29 Declaring an allocator

This is what the allocator might look like:

```
void * C::operator new (size_t nBytes)
{
   Handle   h;
   h = NewHandle(nBytes);
   if (h == NULL) {
      // try to get memory from
      // a secret stash
   }
   return h;
}
```

Example 23-30 Defining an allocator

Since allocators execute before any constructor, you can't do anything with the instance variables or methods of the object. All you can do is allocate memory.

Warning

Be sure that your allocator allocates the right kind of memory. If your class is declared `indirect`, the allocator should allocate handles. If your class is declared `direct`, it should allocate pointers.

If a class and its superclass have allocators, the allocator called is for the class being created.

```
class A {
   void * operator new (size_t);
   ...
};

class B : public a{
   void * operator new (size_t);
   ...
};

f()
{
   A  *p;
   A  *q;

   p = new A;     // uses A::operator new()
   q = new B;     // uses B::operator new()
}
```

Example 23-31 Only the allocator for the class created executes

Deallocators

Deallocators deallocate memory for an existing object. If `C` is the name of a class, the name of its deallocator is `C::operator delete()`. The function `operator delete` takes a pointer to an object as a parameter and returns nothing:

```
class C {
   void operator delete (void *);
   ...
};
```

Example 23-32 Declaring a deallocator

The deallocator might look like this:

```
void C::operator delete (void *theObject)
{
   if (IsPrivateStore(theObject)) {
      // return to private store
   } else
      DisposHandle ((Handle) theObject);
}
```

Example 23-33 Defining a deallocator

Since deallocators execute after destructors, the object that's being deallocated no longer exists. The only thing you can do in a deallocator is return the object's memory to the free store.

If a class and its superclass have deallocators, the deallocator called is for the run-time class of the object being deleted.

```
class A {
   void operator delete (void *p);
   ...
};

class B : public a{
   void operator delete (void *p);
   ...
};

f()
{
   A  *p = new A;
   A  *q = new B;
   ...
   delete p;   // uses A::operator delete()
   delete q;   // uses B::operator delete()
}
```

Example 23-34 Only the deallocator for the class deleted executes

Default allocator and deallocator

If a class does not define an allocator or deallocator, THINK C uses its default built-in ones. If the class is declared `indirect`, or if the Classes Are Indirect by Default option is on, the default allocator calls the Toolbox routine `NewHandleClear()`. If the class is a direct class, the default allocator calls `NewPtrClear()`. The default deallocator calls the Toolbox routine `DisposeHandle()` for indirect classes and `DisposePtr()` for direct classes.

In THINK C you cannot change the default allocator or deallocator. Instead, you can specify an allocator and a deallocator for a root class and declare all classes to be descendants of that class.

Tips and Techniques

This section gives some general tips for making writing object-oriented programs a little easier. It describes how to organize your classes, how to initialize objects easily, how to make sure your

objects behave correctly under the Macintosh Memory Manager, and how the THINK C implementation of objects relates to C++.

Organizing your classes

It's a good idea to isolate classes to make them easy to use and reuse. For each class, create a `.h` file that contains the class declaration and a corresponding `.c` file that contains the method definitions. The names of the files should be the same as the names of the classes.

Your `.h` file should first include the `.h` file for the superclass of the class you're declaring. Next declare your class. Since several classes may try to include your header file, you need a way to keep it from being included more than once. THINK C gives you a way to do this easily.

To prevent a header file from being included more than once, use the `#pragma once` preprocessor directive. This directive tells the compiler that the header file should be included only once for that source file.

Assume you're defining a class called `ClassName`. The file that implements your methods would be called `ClassName.c`. The corresponding header file would be called `ClassName.h`. Here's an example of what `ClassName.h` looks like:

```
#pragma once   // To keep from #including
               // it again.
#include "SuperClass.h"

struct ClassName : SuperClass {
      ...
};
```

Example 23-35 Using #pragma once in a class declaration file

Objects and the Macintosh

See "Object Extension Options" later in this chapter for information about options that affect the object extensions.

The most important thing to keep in mind about THINK C objects is that if they're declared `indirect`, or if the Classes are Indirect by Default option is on, the object is a Macintosh handle. Any reference to an instance variable through an indirect object is implicitly a handle reference. Because of this implementation, your program should not rely on the addresses of instance variables, particularly in calls to Toolbox routines that may move memory.

Look at this hypothetical implementation of a window object:

```
class Window : indirect {
   WindowPtr   theWindow;
   Rect        growRect;
   Rect        dragRect;

public:
      Window(void);  // constructor
      ~Window(void); // destructor

virtual  short    Hit(Point where);
virtual  void     Drag(Point where);
virtual  short    Draw(void);
};
```

This is the way you might implement the `Drag()` method:

```
void Window::Drag(Point where)
{
   DragWindow(theWindow, where, &dragRect);
}
```

The problem with this definition is that it will probably crash the program at some point. `DragWindow()` is a Macintosh Toolbox routine that may move memory, including the handle `this`. By the time `DragWindow()` is invoked, the address of `dragRect` has already been calculated. If `DragWindow()` moves memory, and `this` moves to a different part of memory, the address being used for `dragRect` will actually refer to some other, possibly invalid, data.

A better definition is this:

```
void Window::Drag(Point where)
{
   Rect  dragRect;

   dragRect = this->dragRect;
   DragWindow(theWindow, where, &dragRect);
}
```

In this case, `Drag()` copies the instance variable into a local variable with the same name. Since the address of the local variable never changes within a function block, this use is safe. Though it's not necessary to give the local variable the same name as the instance variable, this shadowing technique keeps you from accidentally using the address of an instance variable.

You should not rely on the order of evaluation in any assignment statement. To be completely safe, program defensively and assume that any operation that can move memory, will move memory.

For example, be careful about passing arrays as arguments to functions that move memory:

```
struct A : indirect {
   Handle theHandles[100];
   void M(void);
};

void MakeHandles(short howMany,
   Handle hArray[])
{
   short i;

   for (i = 0; i < howMany; i++)
      hArray[i] = NewHandle(1024);
}

void A::M(void)
{
   MakeHandles(100, theHandles);
}
```

Example 23-36 Moving memory unexpectedly

Whenever you pass an array as an argument to a function, it is passed by reference. That means that when `A::M()` calls `MakeHandles()`, it gets the address of the first element of `theHandles`. When `MakeHandles()` calls `NewHandle()`, it's likely that the object will move, and at that point, the address that `MakeHandles()` is using is no longer valid.

You can use the Memory Manager routines `HLock()`, `HUnlock()`, `HGetState()`, and `HSetState()` to keep indirect objects from moving around. Most of the time, you won't need to use these

routines if you use the shadowing technique shown above. In the case of arrays, though, your only recourse is to lock the object:

```
void A::M(void)
{
   SignedByte hState;

   hState = HGetState(this);
   MoveHHI(this)
   HLock(this)

   MakeHandles(100, theHandles);

   HSetState(this, hState);
}
```

Example 23-37 Locking an object to keep it from moving

Direct and indirect classes

The only difference between direct and indirect classes is the way in which objects are allocated in memory. Objects of a direct class are allocated as pointers; objects of an indirect class are allocated as handles.

Since handles can move freely in memory, indirect objects are more "memory-friendly." On the other hand, referring to any member of an indirect object needs an extra level of indirection, so indirect objects are slightly slower. Direct objects require only one level of indirection, so they're faster. But direct objects might tend to fragment your heap.

Object Extension Options

This section describes options that control the object extensions to THINK C. For additional information about the options dialog, see Chapter 8, "Organizing Projects."

Turning on the object extensions

For the object extensions to work, you must set the THINK C + Objects option on in the Language Settings page of the **THINK C**

options dialog. Turn on the Language Extensions check box, and select the THINK C + Objects radio button, as shown in Figure 23-1.

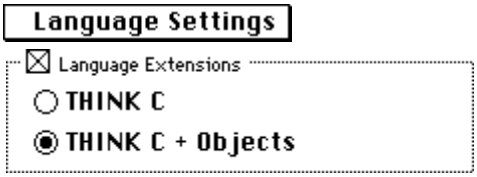

Figure 23-1 THINK C + Objects option

Four more options control how THINK C generates code for programs that use the object extensions. Three are on the Compiler Settings page of the **THINK C options** dialog control, shown in Figure 23-2. The other is in the Preferences page of the **THINK Project Manager options** dialog, shown in Figure 23-3.

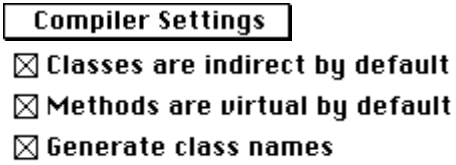

Figure 23-2 THINK C options that affect object extensions

Figure 23-3 Project Manager option that affects object extensions

Classes are indirect by default

With this option on, root classes that you declare with the `class` keyword, and for which you do not provide a storage specifier, are treated as if the storage specifier were `indirect`.

This option doesn't affect classes declared with the `struct` keyword. Classes declared that way must have a storage specifier.

You can still use the storage specifiers `direct` and `indirect` in any root class declaration, regardless of how this option is set. In the example below, class `A` is either `direct` or `indirect`, depending

on how this option is set. Class `B` is always `direct`, regardless of how the option is set.

```
class A {
    member declarations
};

class B : direct {
    member declarations
};
```

Example 23-38 Specifying class storage

Methods are virtual by default

With this option on, every method in a class is implicitly virtual. For maximum compatibility with C++, or to take advantage of nonvirtual methods, be sure this option is off.

Regardless of how this option is set, remember that:

- Static methods are never virtual
- Allocators and deallocators are never virtual
- Constructors are never virtual
- Destructors are always virtual

Generate class names

With this option on, THINK C generates class names in your application, so you can use the functions `new_by_name()` and `class_name()`. If you use very long class names, turning this option off can significantly reduce the size of your global data.

Optimize monomorphic methods

With this option on, the linker optimizes calls to monomorphic methods—virtual methods that are never overridden—so the call does not have to go through the method look-up table. This option affects code written in both THINK C and Symantec C++. You should leave this option on.

Special Functions

This section describes functions that let you create objects by name and test for class membership. These functions are extensions to THINK C that do not appear in standard C++. (They are supported in Symantec C++, but only for objects that derive from Pascal Object.) Be sure you `#include <oops.h>` in the source file that uses these functions.

new_by_name

```
void *new_by_name (char *aName);
```

Creates a new object. `AName` is a C string that names the class. If there is no class with the given name, `new_by_name` returns `NULL`.

The named class must be defined in your program. If your program defines a class but never creates an instance of it, the smart linker may remove the class definition, so you'll be unable to use `new_by_name` to create an instance of that class. One way to work around this restriction is to create a dummy object, then immediately delete it. Another way is to use the `member()` function to refer to the class.

class_name

```
char *class_name (void *anObject);
```

Returns a string that is the name of the class `anObject` belongs to. This function does not check `anObject` to make sure it is a valid object reference.

member

```
char member (void *anObject, void *aClass);
```

Returns `1` if `anObject` is an instance of `aClass` or an instance of one of its ancestors; otherwise returns zero.

Differences Between THINK C and C++

The syntax that THINK C uses for objects is a subset of C++ as described in *The Annotated C++ Reference Manual* by Margaret Ellis and Bjarne Stroustrup (Addison-Wesley). THINK C implements simple objects without multiple inheritance. THINK C does not implement any non-object extensions from C++ such as operator or function overloading.

General differences

The main difference between THINK C and C++ is that, at its core, THINK C is an ANSI C compiler, and there are subtle yet significant differences between ANSI C and C++ in addition to the obvious ones. Keep in mind that C and C++ are completely different languages.

There are a few anomalous cases in which something that works in THINK C might not work in C++, even if you've taken care not to use any of the THINK C extensions. Or there might be cases in which a C++ program that restricts itself to the implementation of objects in THINK C behaves differently in THINK C.

For guidance in these cases, look at section 18.2, "C++ and ANSI C," of *The Annotated C++ Reference Manual*. The scope rules of ANSI C and C++ are significantly different. Refer to section 3.1.2.1, "Scopes of Identifiers," of the ANSI C Standard (*American National Standard for Information Systems—Programming Language—C* ANS X3.159-1989).

Keywords

When the THINK C + Objects language extension option is on, THINK C reserves the following names as keywords:

```
class                delete
new                  virtual
```

The following names are interpreted specially in context, but they are not keywords:

```
direct         indirect       inherited
operator       private        protected
public         this
```

Value objects and reference objects

In THINK C you can refer to an object only through a pointer. C++ lets you write this:

```
class myClass {
   int   someField;
   void  SomeMeth(void);
} aBadObject;
```

Example 23-39 A value object

In THINK C, you have to write it like this:

```
class myClass {
   int   someField;
   void  SomeMeth(void);
} *aGoodObject;

aGoodObject = new myClass;
```

Example 23-40 A reference object

THINK C extensions

THINK C adds some features that do not appear in C++. If you use these features, your program will not port directly to C++.

- The word `inherited` is allowed as a scope qualifier to refer to a class' superclass.
- The `direct` and `indirect` words are allowed as storage specifiers for class declarations.
- The `new_by_name` function can be used for instantiation by name.
- The `class_name` function can be used to get a class name from an object.
- The `method` can be used to test object membership in a class.

Access declarations, friends, and multiple inheritance

THINK C does not support access declarations that allow classes to adjust access to specific members of a superclass. THINK C does not support friends or multiple inheritance.

Constructors and destructors

The only kind of constructor that THINK C supports is a default constructor—that is, a constructor with no arguments. The only way you can invoke a constructor is by creating an object with `new`. You can't create an object by calling its constructor directly, as you can in C++.

In THINK C, destructors are implicitly `virtual`, whether or not they are declared `virtual` even if the Methods Are Virtual by Default option is off.

Scope resolution operator

THINK C does not support the unary scope resolution operator.

Sizeof operator not allowed

You cannot use the `sizeof` operator to take the size of a class name or of an object. This use was allowed in THINK C 4.0.